C14

```c
1    /*
2    ** Copyright 1996,1997 EMC Corporation
3    */

5    /*
6    **
7    ** EDMDispatchConfig.c
8    **
9    ** Mission Statement: These are functions to deal with the config file
                           APIs.
10   **
11   **                    Since the config file can't be handled in a
12   **                    thread safe manner we need to mutex lock around
                            all uses of it.
13   **
14   ** Primary Data Acted On:
15   **
16   ** Compile-Time Options:
17   **
18   ** Basic idea here: Module for config file interaction
     */

20   /*
21   ** The following provides an RCS id in the binary that can be located
22   ** with the what(1) utility.  The intent is to keep this short.
23   */
24   #if !defined(lint)
25   static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchConfig.c,v $ "
26                   "$Revision: 1.23 $ "
27                   "$Date: 1997/02/06 20:49:15 $ " ;
28   #endif

30   /* #define _POSIX_SOURCE       unable to compile with this define set */
31   /* #define _XOPEN_SOURCE       unable to compile with this define set */

33   #include <esl/c_portable.h>
34   #include <esl/ep_xopen.h>
35   #include <esl/inout.h>

37   #include <unistd.h>
38   #include <pthread.h>
39   #include <memory.h>
40   #include <sys/time.h>
41   #include <sys/types.h>

43   #include <ebconfig/rbconfig.h>
44   #include <EDMDispatchConfig.h>

46   static RBC_CONFIGS *rbc = NULL;

48   static boolean_ty  first = TRUE;

50   static pthread_mutex_t    G_configMtx = PTHREAD_MUTEX_INITIALIZER;

52   static void
53   DispatchInitializeConfigMutex()
54   {
55       pthread_mutex_init(&G_configMtx, NULL);
56       first = FALSE;
57   }
```

```c
59   void
60   DispatchReadConfig()
61   {
62       eperrno  ret;

64       if (first == TRUE)
65           DispatchInitializeConfigMutex();

67       pthread_mutex_lock(&G_configMtx);

69       if (rbc != NULL)
70       {
71           rbc_freeconfig(rbc);
72           rbc = NULL;
73       }

75       ret = rbc_parse_config(NULL, &rbc,
76                              RBC_PARSE_DO_NOT_PRESERVE |
                                RBC_PARSE_APPLY);

78       pthread_mutex_unlock(&G_configMtx);
79   }
```

```
81      boolean_ty
82      DispatchCheckRestorePermission(char *host, char *username)
83  1   {
84  1       int       root = 0777;
85  1       eperrno   err;
86  1       boolean_ty allowed;

88  1       // The Configuration structure is not set up correctly !!! STEVE
                                                                        HOWARD

90  1       pthread_mutex_lock(&G_configMtx);

92  1       allowed = rbc_canirecover(rbc, host, username, &root, &err);

94  1       pthread_mutex_unlock(&G_configMtx);

96  1       allowed=1;
97  1       return allowed;
98      }
```

```
 1  /*
 2  **
 3  ** Copyright 1996,1997 EMC Corporation
 4  **
 5  ** EDMDispatchBackground.c
 6  **
 7  **
 8  ** Mission Statement: This is the entry point for the cleanup thread.
 9  **                    Its main purpose is to do some background
                          processing
                          that we don't want to do elsewhere.
10  **
11  **
12  ** Primary Data Acted On:
13  **
14  ** Compile-Time Options:
15  **
16  ** Basic idea here: Module for Background thread.
17  */

19  /*
20  ** The following provides an RCS id in the binary that can be located
21  ** with the what(1) utility. The intent is to keep this short.
22  */
23  #if !defined(lint)
24  static char     RCS_id [] = "@(
                #)$RCSfile: EDMDispatchBackground.c,v $ "
25          "$Revision: 1.23 $ "
26          "$Date: 1997/02/06 20:49:15 $" ;
27  #endif

29  /* #define _POSIX_SOURCE    unable to compile with this define set */
30  /* #define _XOPEN_SOURCE    unable to compile with this define set */

32  #include <esl/c_portable.h>
33  #include <esl/ep_xopen.h>
34  #include <esl/inout.h>

36  #include <unistd.h>
37  #include <pthread.h>
38  #include <memory.h>
39  #include <sys/time.h>
40  #include <sys/types.h>

42  #include <restore/dispatch_daemon.h>
43  #include <EDMDispatchConfig.h>
44  #include <EDMDispatchSession.h>
45  #include <EDMTimedMessageApi.h>
46  #include <EDMDispatchBackground.h>

49  // Number of background activities run from the cleanup thread
50  #define NUMBER_OF_ACTIVITIES  4

51  static const int oneDaySeconds = SECONDS_PER_DAY;     // one day in seconds
52  static const int configSeconds = 30;  // 30 seconds
53  static const int oneMinute = 60; // one minute in seconds

55  // Structure used in the cleanup thread to schedule background
                                                             activities
56  struct Schedule {
57      long    frequency;
58      long    lastrun;
59      long    nextrun;
60      void    (*cleanupfunc)();
61  };
```

```
 63  void *
 64  DispatchBackground(void *buff)
 65  {
 66      time_t      currTime;
 67      time_t      sleepfor = 0;
 68      time_t      difference = 0;

 70      // These are all the activities that are scheduled
 71      //   Frequency,      lastrun,      nextrun,   function to call
 72      struct Schedule sched[NUMBER_OF_ACTIVITIES] = {
 73  configSeconds,
 74      { SECONDS_PER_HOUR,     -1,     -1,     -1,     DispatchReadConfig },
 75      { oneMinute * 5,        -1,     -1,     -1,     CheckDispatchSessions },
 76      { oneMinute,            -1,     -1,     -1,     SendPingMessagesToSession },
 77      { oneMinute,            -1,     -1,     -1,     DrainSessionDescriptors },
                                 -1,     -1,     -1,     ReportLateTimedMessage }
 78      };

 80      // DispatchReadConfig();

 81      // Initialize each elements last and next run.
 82      // The first run will be after sleepfor seconds.
 83      for (int i = 0; i < NUMBER_OF_ACTIVITIES; i++)
 84      {
 85          sched[i].lastrun = time(NULL);
 86          sched[i].nextrun = sched[i].lastrun + sched[i].frequency;
 87          difference = sched[i].nextrun - sched[i].lastrun;

 89          // We need to set the sleepfor value to something on the
 90          // first pass so we have something to compare to. The
 91          // lowest time is what we'll sleep for.
 92          if (i == 0 || difference < sleepfor)
 93              sleepfor = difference;
 94      }

 96      currTime = time(NULL);

 98      // Run things forever
 99      while(1)
100      {
101          // Sleep for the shortest amount of time needed
102          sleep(sleepfor);

104          currTime = time(NULL);

106          // See which activities need to be run on this pass.
107          for (int i = 0; i < NUMBER_OF_ACTIVITIES; i++)
108          {
109              if (sched[i].nextrun <= currTime)
110              {
111                  // This activity needs running. Call the function
112                  // and change the lastrun and next run values.
113                  sched[i].cleanupfunc();
114                  sched[i].lastrun = currTime;
115                  sched[i].nextrun = sched[i].lastrun + sched[i].frequency;
116              }

118              // See how long until this needs to be run.
119              difference = sched[i].nextrun - currTime;

121              // We need to set the sleepfor value to something on the
122              // first pass so we have something to compare to. The
123              // lowest time is what we'll sleep for.
```

```
124   3        if (i == 0 || difference < sleepfor)
125   3            sleepfor = difference;
126   2        }
127   1   } // while (1)

129   1   return buff;
130      }
```

```
 1  /*
 2  ** Copyright 1996,1999 EMC Corporation
 3  */
 5  /*
 6  **
 7  ** EDMDispatchSession.cc
 8  **
 9  ** Mission Statement: This is where all session management occurs.
10  **
11  ** Primary Data Acted On:
12  **
13  ** Compile-Time Options:
14  **
15  **    USE_SUNRPC - Compile source with sunrpc
16  **                 support.  If
17  **                 not set, assume DCE support.
18  **
19  ** Basic idea here: Module for session management
20  */

21  ** The following provides an RCS id in the binary that can be located
22  ** with the what(1) utility.  The intent is to keep this short.
23  **
24  #if !defined(lint)
25  static char    RCS_id [] = "@(#)$RCSfile: EDMDispatchSession.cc,v $ "
26                 "$Revision: 1.23 $ "
27                 "$Date: 1997/02/06 20:49:15 $" ;
28  #endif

30  /* #define _POSIX_SOURCE    unable to compile with this define set */
31  /* #define _XOPEN_SOURCE    unable to compile with this define set */

33  #include <esl/c_portable.h>
34  #include <esl/ep_xopen.h>
35  #include <esl/inout.h>

37  #include <pthread.h>
38  #include <memory.h>
39  #include <sys/time.h>
40  #include <sys/types.h>
41  #include <syslog.h>

43  // Rogue Wave includes
44  #include <rw/collect.h>
45  #include <rw/rwfile.h>
46  #include <rw/vstream.h>
47  #include <rw/bintree.h>

49  #include <csc/csconn.h>
50  #include <restore/dispatch_daemon.h>
51  #include <restore/dispatch_protocol_client.h>
52  #include <EDMSession.h>
53  #include <EDMReturnMessageApi.h>
54  #include <EDMDHandleMgrApi.h>
55  #include <EDMDispatchSession.h>
56  #include <EDMDispatchConfig.h>
57  #include <EDMDcr_rstsvc.h>

59  #include <EDMDispatchLog.h>

61  static RWBinaryTree        G_sessionTree;

63  static pthread_mutex_t     G_sessionTreeMtx = PTHREAD_MUTEX_INITIALIZER;
64  extern ElinkHandlePtr_ty   ElinkHandle;
```

```
66  static int maxDisconnectTime = SECONDS_PER_HOUR;   // one hour

68  /*********************************************************************
69  **
70  ** Routine:   LockSessionMutex
71  **
72  ** Inputs:    None
73  **
74  ** Outputs:   None
75  **
76  ** Return Codes:
77  **            None
78  **
79  ** Purpose:   Lock the session mutex.
80  **
81  *********************************************************************
82  */

84  static void
85  LockSessionMutex()
86  {
87      static boolean_ty first = TRUE;

89      if (first == TRUE)
90      {
91          first = FALSE;
92          pthread_mutex_init(&G_sessionTreeMtx, NULL);
93      }

95      pthread_mutex_lock(&G_sessionTreeMtx);
96  }
```

```
 98   /*********************************************************************
 99   **
100   **  Routine:   UnlockSessionMutex
101   **
102   **  Inputs:    None
103   **
104   **  Outputs:   None
105   **
106   **  Return Codes:
107   **             None
108   **
109   **  Purpose:   Unlock the mutex for the session tree object
110   **
111   **
112   *********************************************************************
114   static void
115   UnlockSessionMutex()
116 1 {
117 1    pthread_mutex_unlock(&G_sessionTreeMtx);
118   }
```

```
120   /*********************************************************************
121   **
122   **  Routine:   InitializeSession
123   **
124   **  Inputs:    DD_initialize_args *arg - args sent via RPC for starting
                                              session
125   **             struct svc_req *req - the request block from RPC
126   **
127   **  Outputs:   DD_initialize_result *res - the result structure which
                                                tells whether
128 3             operation succeeded or failed.
129   **
130   **  Return Codes:
131   **             None
132   **
133   **  Purpose:   Initialize a session for the GUI.
134   **
135   *********************************************************************
136   */
138   void
139   InitializeSession(IN DD_initialize_args *arg, IN struct svc_req *req,
140                     OUT DD_initialize_result *res)
141 1 {
142 1    EDMSession    *session;
143 1    EDMSession    *ret;
144 1    pthread_t      id;
145 1    time_t         t;

147 1    if (arg == NULL || req == NULL || res == NULL)
148 2    {
149 2       return;
150 1    }

152 1    t = time(NULL);

154 1    session = new EDMSession();

156 1    if (session == NULL)
157 2    {
158 2       res -> status = DD_SERVICE_FAILURE_NONEXEC;
159 2       return;
160 1    }

162 1    session -> initSession();

164 1    session -> setStartTime(t);

166 1    session -> setOperationType(arg -> service);

168 1    session -> setStatus(DD_SERVICE_STARTING);

170 1    if (arg -> username != NULL && arg -> hostname != NULL)
171 2    {
172 2       switch(arg -> service)
173 3       {
174 3          // code is commented out because we do not
175 3          // want to read tthe config for permission information
176 3          // at this time, it is a waste of cycles
177 3   #if 0
178 3          case DD_SERVICE_RESTORE : boolean_ty allowed;
180 3             allowed =
```

```
           DispatchCheckRestorePermission(
               arg->hostname,
               arg -> username);

           if (!allowed)
           {
               res -> status =
                   DD_SERVICE_FAILURE_PERMS;
               delete session;
               return;
           }

           break;

#endif
       default: // Add some error messsage for unknown service

           break;
       };
   }
   else
   {
       res -> status = DD_SERVICE_FAILURE_NONEXEC;
       delete session;
       return;
   }
   LockSessionMutex();

   ret = (EDMSession *) G_sessionTree.insert((
                 RWCollectable *) session);

   UnlockSessionMutex();

   if (ret == NULL)
   {
       res -> status = DD_SERVICE_FAILURE_NONEXEC;
       delete session;
       return;
   }

   session -> getSessionID(&res -> service -> service_handle);

   // Call Steve's thread
   pthread_create(&id, NULL, &DDRSTsvc_init, (void *) session);

   session -> setThreadID(id);

   return;
}
```

```
/*************************************************************
**
** Routine:     SendPingMessagesToSession
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**                  None
**
** Purpose:     Queue up all the ping messages to the sessions.  If they don't
                 respond they should be considered dead.
**
**
**
*************************************************************/

void
SendPingMessagesToSession()
{
   EDMSession *sess;

   LockSessionMutex();

   RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                 G_sessionTree);

   while ( sessionIterator != NULL &&
          (sess = (EDMSession*) (*sessionIterator)()) != NULL )
   {
       DD_client_session_id sid;
       rpc_binding_handle_t *cscb = NULL;
       int                  status;
       int                  ret;

       if (sess -> getStatus() != DD_SERVICE_RUNNING)
           continue;

       sess -> getSessionID(&sid);

       ret = GetCSHandle(&sid, &cscb, &status);

       if (ret != 0 || cscb == NULL || *cscb == NULL)
           continue;

       PushResponseMessage(dp_ping_request, sid, cscb, &status);
   }

   // through with iterator
   if (sessionIterator != NULL)
   {
       delete sessionIterator;
   }

   UnlockSessionMutex();
}
```

```
282
283   }
284   **
285   /*****************************************************************
286   **
287   **  Routine:  UpdateSessionLastReceived
288   **
289   **  Inputs:   DD_client_session_id *sessID - session that sent us
290   **                                           something
291   **
292   **  Outputs:  None
293   **
294   **  Return Codes:
295   **     0 on success and non-zero otherwise
296   **
297   **  Purpose:  Update the specified session with the lastest received
      **            message
      **            time.
      **
      **
      *****************************************************************
299   */
300   int
301   UpdateSessionLastReceived(DD_client_session_id *sessID)
302   {
303      time_t   last = time(NULL);
304      EDMSession  *session;
306      EDMSession  *ret;

308      session = new EDMSession();

310      if (session == NULL)
311      {
312         EDMDispatch_logent(
313            __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
315            "Failure to create a session block");

317         return -1;
319      }

321      session -> setSessionID(sessID);

323      LockSessionMutex();

325      ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

327      UnlockSessionMutex();

329      delete session;

331      if (ret == NULL)
333      {
335         EDMDispatch_logent(
336            __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
               "Failure to update session %ld:%ld received time",
               sessID -> high, sessID -> low);

            return -1;
         }

         ret -> setLastReceived(last);

         return 0;
      }
```

```
338   }
      /*****************************************************************
339   **
340   **  Routine:  UpdateSessionLastSent
341   **
342   **  Inputs:   DD_client_session_id *sessID - session that sent us
343   **                                           something
344   **
345   **  Outputs:  None
346   **
347   **  Return Codes:
348   **     0 on success and non-zero otherwise
349   **
350   **  Purpose:  Update the specified session with the lastest sent
351   **            message
352   **            time.
353   **
      *****************************************************************
      */
355   int
356   UpdateSessionLastSent(DD_client_session_id *sessID)
357   {
358      time_t   last = time(NULL);
359      EDMSession  *session;
360      EDMSession  *ret;

362      session = new EDMSession();

364      if (session == NULL)
365      {
366         EDMDispatch_logent(
367            __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
368            "Failure to create a session block");

369         return -1;
371      }

373      session -> setSessionID(sessID);

375      LockSessionMutex();

377      ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

379      UnlockSessionMutex();

381      delete session;

383      if (ret == NULL)
384      {
385         EDMDispatch_logent(
386            __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
387            "Failure to update session %ld:%ld sent time",
389            sessID -> high, sessID -> low);

391         return -1;
         }

         ret -> setLastSent(last);

         return 0;
392   }
```

```
394   /*****************************************************************
395                                                               *******
396   **                                                          *******
397   **
398   **  Routine:    CheckDispatchSessions
399   **
400   **  Inputs:    None
401   **
402   **  Outputs:   None
403   **
404   **  Return Codes:
405   **             None
406   **
407   **  Purpose: Look for dead sessions and kill them off
408   **
              *****************************************************************
              ********
410   */
411
      void
      CheckDispatchSessions()
412   {
413       EDMSession    *sess;
414       int           status = 0;
415       int           ret = 0;
416       time_t        currTime;
417       RWBinaryTree  reaperTree;
419       currTime = time(NULL);
421       LockSessionMutex();
423       RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                        G_sessionTree);
425       while ( sessionIterator != NULL &&
426             (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {
428   2       if ( (sess->getLastReceived() != 0) ||
429   2             (sess->getStartTime() <= currTime - maxDisconnectTime &&
430   2             (sess -> getStatus
                    ) == DD_SERVICE_FAILURE_NONEXEC || sess -> getStatus (
              <= currTime - maxDisconnectTime && sess->getLastReceived() != 0) ||
431   2             sess -> getStatus() == DD_SERVICE_FAILURE_EXEC ||
                        ) == DD_SERVICE_FAILURE_PERMS) ) {
432   2       {
433   3           // Insert it into the reaper tree
434   3           (void) reaperTree.insert(sess);
435   2       }
436   1     }
438   1     // through with iterator
439   1     if (sessionIterator != NULL)
440   2     {
441   2         delete sessionIterator;
442   1     }
444   1     UnlockSessionMutex();
446   1     // If the reaper tree has something in it then use those entries
447   1     // things from the query tree.
448   1     if (reaperTree.entries() > 0)
449   2     {
450   2         sessionIterator = new RWBinaryTreeIterator(reaperTree);
```

```
452   2       while ( sessionIterator != NULL &&
453   3             (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {
454   3           DD_client_session_id  sessID;
456   3           sess -> getSessionID(&sessID);
458   3           ret = removeSession(&sessID, &status);
460   3           if (ret != 0)
461   4           {
462   4               EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
463   4                   "Failure to remove session %ld:%ld",
464   4                   sessID.high, sessID.low);
465   4               continue;
466   3           }
467   3           else
468   4           {
469   4               EDMDispatch_logent(
470   4                   __FILE__, __LINE__, LOG_INFO, 0, 0,
471   4                   "Removing session %ld:%ld,
                        Haven't recieved anything since %ld. Current %ld",
472   4                   sessID.high, sessID.low,
473   4                   sess -> getLastReceived(),
                        currTime - maxDisconnectTime);
475   3           }
477   3           ret = deleteHandleSet(&sessID, &ELinkHandle, &status);
478   4           if (ret != 0)
479   4           {
480   4               EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
481   4                   "Failure to delete handles for
                        session %ld:%ld",
482   3                   sessID.high, sessID.low);
483   2           }
485   2       }
486   2       // through with iterator
487   3       if (sessionIterator != NULL)
488   3       {
489   2           delete sessionIterator;
491   2       }
492   1       reaperTree.clear();
493       }
```

```
495   1   /*************************************************************
496
497       **
498       ** Routine:  DrainSessionDescriptors
499       **
500       ** Inputs:   None
501       **
502       ** Outputs:  None
503       **
504       ** Return Codes:  None
505       **
506       ** Purpose:  Drain whatever data is on stdout and stderr for sessions.
507       **
508       **
509           *************************************************************

          */

511       void
512       DrainSessionDescriptors()
513   1   {
514   1       int hout = 0, herr = 0, status = 0;
515   1       int selret = 0;
516   1       int i = 0;
517   1       char buff[1024];
518   2       struct timeval timetowait = {
519   2           1, 0
520   1       };
521   1       fd_set stdoutSet;
522   1       fd_set stderrSet;

524   1       getStdoutSet(&stdoutSet, &hout, &status);

526   1       if ( (selret = select(
527   2               hout + 1, &stdoutSet, NULL, NULL, &timetowait)) >= 0)
528   2       {
529   3           for (; i < hout+1; i++)
530   3           {
531   4               if (FD_ISSET(i, &stdoutSet))
532   4               {
533   4                   while (read(i, buff, 1024) > 0);
534   2               }
535   1           }
                 }
537   1       getStderrSet(&stderrSet, &herr, &status);

539   1       if ( (selret = select(
540   2               herr + 1, &stderrSet, NULL, NULL, &timetowait)) >= 0)
541   2       {
542   3           for (i = 0; i < herr+1; i++)
543   4           {
544   4               if (FD_ISSET(i, &stderrSet))
545   4               {
546   4                   while (read(i, buff, 1024) > 0);
547   2               }
548   1           }
549   1       }
          }
```

```
551   1   /*************************************************************

553       **
554       ** Routine:  GetSessionStatus
555       **
556       ** Inputs:   DD_client_session_id *ssid - session ID to check the
                                                    status of
557       **
558       ** Outputs:  int *status - status of the function call
559       **            int *s_status - session status
560       **
561       ** Return Codes:
562       **            0 if successful and non-zero otherwise
563       **
564       ** Purpose:  Get status on the session.
565       **
566           *************************************************************
567
568       */
          int
          GetSessionStatus(
569   1       DD_client_session_id *ssid, int *s_status, int *status)
570   1   {
571   1       EDMSession  *sess;
             EDMSession  *ret;

573   1       if (status == NULL)
574   2       {
575   2           return -1;
576   1       }

578   1       if (ssid == NULL || s_status == NULL)
579   2       {
580   2           *status = SESSION_BAD_ARGS;
582   2           return -1;
583   1       }

585   1       sess = new EDMSession();

587   1       if (sess == NULL)
588   2       {
589   2           EDMDispatch_logent(
590   2               __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
591   2               "Failure to create a session block");

593   2           *status = SESSION_NO_MEMORY;
594   1           return -1;
             }

596   1       LockSessionMutex();

598   1       sess -> setSessionID(ssid);

600   1       ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

602   1       UnlockSessionMutex();

604   1       delete sess;

606   1       if (ret == NULL)
607   2       {
608   2           EDMDispatch_logent(
609   2               __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                     "Failure to lookup session %ld:%ld",
```

```
610  2
611  2               ssid -> high, ssid -> low);
612  1      return -1;
613  1    }

615  1    *s_status = ret -> getStatus();

617  1    return 0;
618      }
```

`*status = SESSION_LOOKUP_FAILED;`

```
620  /**********************************************************************
621  **                                                              ********
622  **  Routine:   GetDispatchStatus
623  **
624  **  Inputs:    DD_getservicestatus_args *arg - session ID to check the
625  **                                             status of
626  **  Outputs:   DD_getservicestatus_result *res - the result structure
627  **                                               which tells
628  **                             whether operation succeeded or failed.
629  **  Return Codes:
630  **        None
631  **
632  **  Purpose:   Get status on the starting session.
633  **
634  ***********************************************************************
635  */

637  void
638  GetDispatchStatus(IN DD_getservicestatus_args *arg,
                       OUT DD_getservicestatus_result *res)
639  {
640  1    EDMSession   *sess;
641  1    EDMSession   *ret;
642  1    EDMSession   *ret;
643  1    static char buff[CONNECT_HANDLE_SIZE];

645  1    sess = new EDMSession();

647  1    if (sess == NULL)
648  2    { // Give an error
649  2      EDMDispatch_logent(
650  2          __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0, 0,
651  2          "Failure to create a session block");
652  1      return;
         }

654  1    sess -> setSessionID(arg -> service_handle);

656  1    LockSessionMutex();

658  1    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

660  1    UnlockSessionMutex();

662  1    delete sess;

664  1    if (ret == NULL)
665  2    {
666  2      EDMDispatch_logent(
667  2          __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
668  2          "Failure to lookup session %ld:%ld",
670  2          arg -> service_handle.high,
671  2          arg -> service_handle.low);
672  1      res -> status = DD_SERVICE_FAILURE_NONEXEC;
674  1      return;
         }

676  1    memset(buff, 0, sizeof(buff));
```

```
678  1        if (res -> status == DD_SERVICE_RUNNING)
679  2        {
680  2            res -> handle.handle.handle_val = (char *) ret -> getConnectionHandle(
                                                                                        );
681  2            res -> handle.handle.handle_len = CONNECT_HANDLE_SIZE;
682  1        }
     else
     {
684  2            res -> handle.handle_val = (char *) buff;
685  2            res -> handle.handle_len = CONNECT_HANDLE_SIZE;
686  2        }
687  1    }
688  1 }
```

```
690     /***********************************************************************
691     **
692     ** Routine:   GetDispatchInfo
693     **
694     ** Inputs:    DD_getservicestatus_args *arg - session ID to check the
695                      status of.
696     ** Outputs:   SessionBlock *res - the information regarding the
697                      specified session
698     **
699     ** Return Codes:
700     **     None
701     **
702     ** Purpose:  Get status on all the sessions.
703     **
704     ***********************************************************************
        ***********
        */
708     void
        GetDispatchInfo(IN DD_getservicestatus_args *arg,
                        OUT SessionBlock *res)
        {
709  1      EDMSession    *sess;
710  1      EDMSession    *ret;
711  1      SessionInfo   *sinfo, *slast;
712  1      static char   buff[CONNECT_HANDLE_SIZE];
713  1
715  1      LockSessionMutex();
717  1      if (arg -> service_handle.high != 0 && arg -> service_handle.
                                                              low != 0)
718  2      {
719  2          // Looking for a single session. Do a find.
720  2          sess = new EDMSession();
722  2          if (sess == NULL)
723  3          { // Give an error
724  3              EDMDispatch_logent(
                        FILE__, LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                        "Failure to create a session block");
725  3              UnlockSessionMutex();
727  3              return;
728  3          }
729  2
731  2          sess -> setSessionID(&arg -> service_handle);
733  2          ret = (EDMSession *) G_sessionTree.find(sess);
735  2          delete sess;
737  2          if (ret == NULL)
738  3          {
739  3              EDMDispatch_logent(
                        FILE__, LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                        "Failure to lookup session %ld:%ld",
                        arg -> service_handle.high,
                        arg -> service_handle.low);
740  3
741  3              UnlockSessionMutex();
742  3              return;
743  3          }
744  2      }
746  2      res -> totalsessions = 1;
```

```
748  2        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

750  2        if (res -> sess == NULL)
751  3        {
752  3            EDMDispatch_logent(
753  3                __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
754  3                "Failure to allocate session info
755  3                block");

756  2            UnlockSessionMutex();

758  2            return;
             }

760  2        sinfo = res -> sess;

761  2        sinfo -> status = ret -> getStatus();
762  2        sinfo -> jobstarttime = ret -> getStartTime();
763  2        sinfo -> operation_type = ret -> getOperationType();
764  2        sinfo -> lastSent = ret -> getLastSent();
765  2        sinfo -> lastReceived = ret -> getLastReceived();
766  1    }
767  1    else
768  2    {
769  2        res -> totalsessions = 0;

771  2        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

773  2        if (res -> sess == NULL)
774  3        {
775  3            EDMDispatch_logent(
776  3                __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
777  3                "Failure to allocate session info
778  3                block");

779  2            UnlockSessionMutex();

781  2            return;
783  2        }

785  2        RWBinaryTreeIterator *sessionIterator = new
                       RWBinaryTreeIterator(G_sessionTree);

787  2        boolean_ty addnext = FALSE;

789  3        while ( sessionIterator != NULL && (ret = (EDMSession*)
                     (*sessionIterator)()) != NULL )
791  3        {
792  4            int             status;

793  4            if (addnext)
795  4            {
796  5                sinfo -> next = (SessionInfo *) calloc(1, sizeof(
                             SessionInfo));
797  5                if (sinfo -> next == NULL)
                     {
798  6                    break;
                     }
800  4                sinfo = sinfo -> next;
801  3            }

803  3            ret -> getSessionID(&sinfo -> service_handle);
804  3            sinfo -> status = ret -> getStatus();
805  3            sinfo -> jobstarttime = ret -> getStartTime();
```

```
806  3            sinfo -> operation_type = ret -> getOperationType();
807  3            sinfo -> lastSent = ret -> getLastSent();
808  3            sinfo -> lastReceived = ret -> getLastReceived();

810  3            getHandleSet(
811  3                &sinfo -> service_handle, &sinfo -> outhandle,
813  3                &sinfo -> errhandle, &status);

815  3            res -> totalsessions++;

816  3            sinfo -> next = NULL;
817  2            addnext = TRUE;
819  2        }
820  2        // through with iterator
821  3        if (sessionIterator != NULL)
822  3        {
823  2            delete sessionIterator;
825  1        }

827  1        UnlockSessionMutex();
828      }
```

```
830   /*************************************************************
831   **
832   ** Routine:   removeSession
833   **
834   ** Inputs:
835   **
836   ** Outputs:
837   **
838   ** Return Codes:
839   **     None
840   **
841   ** Purpose: Remove the active session object between the GUI and the
842   **          Service.
843   **
844   ** *********************************************************
      */
846   int
847   removeSession(IN DD_client_session_id *sess_id,
848                 OUT int *status)
849   {
850       EDMSession *sess;
851       EDMSession *ret;

853       if (status == NULL)
854       {
855           return -1;
856       }

858       if (sess_id == NULL)
859       {
860           *status = SESSION_BAD_ARGS;
861           return -1;
862       }

864       *status = 0;
865       if (G_sessionTree.isEmpty())
866       {
867           EDMDispatch_logent(
868               __FILE__, __LINE__, LOG_ERR, SESSION_LIST_EMPTY, 0,
869               "No sessions in list.  Can't remove session <%ld:%ld>",
870               sess_id -> high, sess_id -> low);

871           *status = SESSION_LIST_EMPTY;
872           return -1;
874       }

      sess = new EDMSession();

876       if (sess == NULL)
877       {
878           EDMDispatch_logent(
879               __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
880               "Failure to create a session block");

881           *status = SESSION_NO_MEMORY;
882           return -1;
          }

884   sess -> setSessionID(sess_id);

886   LockSessionMutex();

888   ret = (EDMSession *) G_sessionTree.remove(sess);
```

```
890   UnlockSessionMutex();

892   if (ret == NULL)
893   {
894       EDMDispatch_logent(
895           __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
896           "Failure to remove session %ld:%ld",
897           sess_id -> high, sess_id -> low);

898       *status = SESSION_LOOKUP_FAILED;
899       return -1;
900       delete sess;
      }

902   delete ret;
903   delete sess;

905   return 0;
906   }
```

```c
1    /*
2    ** Copyright 1996,1998 EMC Corporation
3    */
5
6    /*
7    ** EDMDispatchLog.c
8    **
9    ** Mission Statement: This is the logging wrapper around esl logging.
10   **
11   ** Primary Data Acted On:
12   **
13   ** Compile-Time Options:
14   **
15   ** Basic idea here: Module for logging
16   */
17
18   /*
19   ** The following provides an RCS id in the binary that can be located
20   ** with the what(1) utility. The intent is to keep this short.
21   */
22   #if !defined(lint)
23   static char    RCS_id [] = "@(#)$RCSfile: bamlogging.c,v $ "
24                              "$Revision: 1.23 $ "
25                              "$Date: 1997/02/06 20:49:15 $ ";
26   #endif
27
28   #include <esl/c_portable.h>
29   #include <esl/ep_xopen.h>
30   #include <esl/inout.h>
31   #include <stdio.h>
32   #include <pthread.h>
33   #include <stdarg.h>
34   #include <string.h>
36   #include <logging/logging.h>
37   #include <EDMmain.h>
38   #include <EDMDispatchLog.h>
40   /***************************************************************
41   **
42   ** Routine: EDMDispatch_logent
43   **
44   ** Inputs:
45   **           file name      -  c source file name
46   **           line number    -  line number in c source file
47   **           priority       -  esl logent priority
48   **           message #      -  esl logent message number
49   **           errno          -  errno optional
50   **           msg format     -  format for printing,
                                    ie "text %s %d..."
51   **           variable args  -  variable arguments for printing
52   **
53   ** Outputs: Calls esl_logent for output.
54   **
55   ** Return Codes:
56   **           None
57   **
58   ** Purpose:   All messages issued from the band should call this
                              routine
59   **           so that they can be consistently formatted. It is
60   **           anticipated that this routine will also be used for
61   **           statistics gathering on messages and perhaps even
62   **           debugging.
63   **
     ** Intended caller: internal only.
```

```c
64   ** ************************************************************
65   */
66
68   void
69   EDMDispatch_logent( IN char   *filename,
70                       IN int     linenum,
71                       IN int     priority,   /* esl_logent priority */
72                       IN int     msg_no,     /* esl_logent message number */
73                       IN int     err_no,     /* errno */
74                       IN char   *msg,        /* msg format for sprintf... */
75                       IN ...                 /* Remaining arguments to msg */
76                     )
78   {
     #define EDMDispatch_MSGBUFF 2048
80       char    tmpbuff[ EDMDispatch_MSGBUFF ];
81       char    msgbuff[ EDMDispatch_MSGBUFF ];
82       va_list vap;                           /* Variable argument pointer */
83       static  pthread_mutex_t  log_mutex = PTHREAD_MUTEX_INITIALIZER;
84       static  boolean_ty       first = TRUE;
86       if (first)
87       {
88           first=FALSE;
89           pthread_mutex_init(&log_mutex, NULL);
90       }
92       if (msg == NULL || filename == NULL)
93           return;
95       /* Setup variable argument list processing */
97       va_start(vap, msg);
99       tmpbuff[ 0 ] = 0;
100      msgbuff[ 0 ] = 0;
103      /* Build caller message with all arguments. */
105      (void) vsprintf( tmpbuff, msg, vap );
107      if ( err_no > 0 )
108      {
109          char *str = strerror(err_no);
111          if (str == NULL)
112              str = "Unknown Error";
114          (void) sprintf(
115                          msgbuff,
116                          "(%s-%d) %s %s <%d>",
117                          filename,
118                          linenum,
119                          tmpbuff,
120                          str,
121                          err_no
                           );
         }
122      else
123          (void) sprintf(
124                          msgbuff,
125                          "(%s-%d) %s",
                            filename,
```

```
126  1                    linenum,
127  1                    tmpbuff );

129  1    /*
130  1     * Use a mutex lock because esl_logent is NOT thread safe.
131  1     */
132  1    pthread_mutex_lock(&log_mutex);

134  1    (void) esl_logent( priority, EB, EDMDISPATCH, msg_no, msgbuff );

136  1    pthread_mutex_unlock(&log_mutex);
137  1  } /* End of EDMDispatch_logent() */
```

```
 1  /*
 2  ** Copyright 1996,1997 EMC Corporation
 3  */

 6  /* EDMDispatchService.c
 7   *
 8   *
 9   * Mission Statement:   RPC entry points.
10   *
11   * Primary Data Acted On:
12   *
13   * Compile-Time Options:
14   *
15   * Basic idea here:
16   *
    */
18  #if !defined(lint)
19  static char   RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $ "
20                            "$Revision: 1.0 $ "
21                            "$Date: 1997/02/06 20:49:15 $ " ;
22  #endif

24  #include <esl/c_portable.h>
25  #include <esl/inout.h>

27  #include <logging/logging.h>
28  #include <csc/cscomm.h>

30  #include <restore/csc_EDMDispatch.h>
31  #include <restore/dispatch_daemon.h>

33  #include <EDMDispatchLog.h>
34  #include <EDMDispatchSession.h>

36  /*
37   * These are all the rpc entry points for the dispatch daemon.
38   * The dispatch daemon is multi-threaded and it is the main thread
39   * which handles all incoming RPC. ONC RPC is single threaded
40   * so each call blocks other RPC calls. This provides us some
41   * safety in the way we handle our data and limits our exposure
42   * to unexpected multithreading problems.
43   */
44  static void FreeSessionInfo(SessionInfo *);

46  /*********************************************************************
47   *******
48  ** Routine:    dd_initialize_1
49  **
50  ** Inputs:     DD_initialize_args * - args for the restore initialize
                   call
51  **
52  ** Outputs:    None
53  **
54  ** Return Codes:
55  **      DD_initialize_result * - result of init function call
56  **
57  ** Purpose:    Function to create a restore session.
58  **
59  ** Intended caller:  Internal Only.
60  ***********************************
61  */

63  DD_initialize_result *
```

```
64  dd_initialize_1_svc(
            IN DD_initialize_args *arg, IN struct svc_req *req )
65  {
66      static DD_initialize_result argzz;

68      InitializeSession(arg, req, &argzz);

70      return &argzz;
71  }
```

```
 73   /*********************************************************************
 74   **
 75   ** Routine:   dd_getservicestatus_1
 76   **
 77   ** Inputs:    DD_getservicestatus_args * - args for the getservicestatus
                                                 call
 78   **
 79   ** Outputs:   None
 80   **
 81   ** Return Codes:
 82   **     DD_getservicestatus_result * - result of status function
                                           call
 83   **
 84   ** Purpose:   Function to poll for status on a session.
 85   **
 86   ** Intended caller:  Internal Only.
 87   **
 88   *********************************************************************
 89   */
 90   DD_getservicestatus_result *
 91   dd_getservicestatus_1_svc(
          IN DD_getservicestatus_args *arg, IN struct svc_req *req )
 92 1 {
 93 1    static DD_getservicestatus_result argzz;

 95 1    GetDispatchStatus(arg, &argzz);

 97 1    return &argzz;
 98   }
```

```
100   /*********************************************************************
101   **
102   ** Routine:   dd_getsessioninfo_1
103   **
104   ** Inputs:    DD_getservicestatus_args * - args for the getsessioninfo
                                                 call
105   **
106   ** Outputs:   None
107   **
108   ** Return Codes:
109   **     SessionBlock * - result of session info call
110   **
111   ** Purpose:   Function to get information on all sessions.
112   **
113   ** Intended caller:  Internal Only.
114   **
115   *********************************************************************
116   */
117   SessionBlock *
118   dd_getsessioninfo_1_svc(
          IN DD_getservicestatus_args *arg, IN struct svc_req *req )
119 1 {
120 1    static SessionBlock argzz;
121 1    static boolean_ty first = TRUE;

123 1    if (first)
124 2    {
125 2       memset(&argzz, 0, sizeof(argzz));
126 2       first = FALSE;
127 1    }
128 1    else
129 2    {
130 2       FreeSessionInfo(argzz.sess);
131 2       argzz.sess = NULL;
132 1    }

134 1    GetDispatchInfo(arg, &argzz);

136 1    return &argzz;
137   }
```

```
139  /************************************************************************
140  **                                                                    *
141  ** Routine:  FreeSessionInfo                                          *
142  **                                                                    *
143  ** Inputs:   SessionInfo * - arg to free                             *
144  **                                                                    *
145  ** Outputs:  None                                                     *
146  **                                                                    *
147  ** Return Codes:                                                      *
148  **           None                                                     *
149  **                                                                    *
150  ** Purpose:  Function to free all SessionInfo structures in a list.  *
151  **                                                                    *
152  ** Intended caller:  Internal Only.                                   *
153  ************************************************************************/
155  static void FreeSessionInfo(SessionInfo *sess)
156 1 {
157 1     if (sess == NULL)
158 1         return;
160 1     if (sess -> next != NULL)
161 1         FreeSessionInfo(sess -> next);
163 1     free(sess);
164 1 }
```